Visual Programming of XSLT from examples

José Paulo Leal[1] and Ricardo Queirós[2]

[1] DCC/FCUP & CRACS,
zp@dcc.fc.up.pt

[2] DI/ESEIG & CRACS,
ricardo.queiros@eu.ipp.pt

Abstract. Vishnu is a tool for XSLT visual programming in Eclipse - a popular and extensible integrated development environment. Rather than writing the XSLT transformations, the programmer loads or edits two document instances, a source document and its corresponding target document, and pairs texts between then by drawing lines over the documents. This form of XSLT programming is intended for simple transformations between related document types, such as HTML formatting or conversion among similar formats. Complex XSLT programs involving, for instance, recursive templates or second order transformations are out of the scope of Vishnu. We present the architecture of Vishnu composed by a graphical editor and a programming engine. The editor is an Eclipse plug-in where the programmer loads and edits document examples and pairs their content using graphical primitives. The programming engine receives the data collected by the editor and produces an XSLT program. The design of the engine and the process of creation of an XSLT program from examples are also detailed. It starts with the generation of an initial transformation that maps source document to the target document. This transformation is fed to a rewrite process where each step produces a refined version of the transformation. Finally, the transformation is simplified before being presented to the programmer for further editing.

Keywords: XML, XSL Transformations, Second order.

1 Introduction

Computer programs are texts written in a programming language. They are an interrelated set of instructions that a computer must execute autonomously. The relationships among instructions are expressed textually using identifiers. Visual programming languages provide an alternative to the use of textual identifiers by using graphical elements to represent structure and connect related parts. The goal of visual programming is to support non-textual interaction to help novice programmers and/or to increase productivity.

The majority of the "visual" programming environments focus only for the creation of graphical interfaces, using text editors for programming the logic of programs. In fact, truly visual programming environments were not successfully implemented for the general-purpose languages such as Java or C + +. They are mostly used for introductory programming languages used by children or non-programmers. They are sometimes used for domain specific languages and languages for graphical interface programming, as is arguably the case of XSLT.

The aim of this paper is the design of a visual programming tool for XSLT. This language was designed for XML style sheet definition, but has won a leading role as a tool for conversion among document types in this formalism. The visual programming tool is based on examples. The programmer provides instances of the source document and the corresponding target document and pairs corresponding texts in both documents using graphical primitives. This data is supplied to a generator that creates an initial XSLT program. Second order XSLT transformations refine it to produce a more structured and generic XSLT program.

This visual approach of XSLT programming has obvious limitations. Only a subset of all possible XSLT transformations is programmable by pairing texts on a source and target documents. For instance, second order transformations or recursive templates are out of its scope. Use cases for Vishnu are formatting XML documents in XHTML and conversion among similar formats. For instance, creating an XHTML view of an RSS feed and converting metadata among several XML formats are among the possible uses of Vishnu. Moreover, we do not expect that the automated features of Vishnu to produce the final version of an XSLT program. We view its final result as a skeleton of a transformation that can be further refined using other tools already available in Eclipse.

The remainder of this paper is organized as follows: section 2 explores some of the related work in this area. In the following section we present the design of a visual programming environment for XSLT, more precisely, its architecture and its internal components used to produce and refine an XSLT program. Finally, we conclude with a summary of the major contributions of this paper and a prospect of future work.

2 Related work

There are several environments for programming in XSLT. Usually these tools are integrated in XML IDE's or in general purpose IDE's such as Eclipse. In the former we can highlight StyleVision and Stylus Studio. StyleVision [1] is a commercial visual stylesheet designer for transforming XML. It allows drag and dropping XML data elements within an WYSIWYG interface. An XSLT stylesheet is automatically generated and can be previewed using the FOP built-in browser. Stylus Studio's [2] is another commercial XML IDE that includes a WYSIWYG XSLT designer. The edition process is guided by simple drag-and-drop operations without requiring prior knowledge of XSLT.

There are also several plug-ins for Eclipse for editing XSLT and the Tiger XSLT Mapper [3] is the most prominent. It is a simple development environment that supports automatic mappings between XML structures and can be edited using the drag-and-drop visual interface. While the mappings and XML structures are modified, the XSLT template is automatically generated and modified. Other examples of Eclipse plug-ins address the XSLT edition [4,5,6] and the XSLT execution [7,8].

There are other tools analogue to Vishnu that are not integrated into Eclipse, as the dexter-xsl [9] which is intended to be used from the command line, the VXT [10] a visual programming language for the specification of XML transformations in an interactive environment and FOA [11] an XSL-FO graphical authoring tool to create XSL-FO stylesheets. It includes a tree visualization scheme to represent the source XML document and the target FO tree structure. FOA generates an XSLT stylesheet that transforms XML content into an XSL-FO document.

Despite the existence of several environments for programming in XSLT, usually integrated into IDE's, they do not use visual editing for programming. Moreover, as far as we know, none of the graphical XSLT programming environment generates programs from examples.

3 The Vishnu application

In this section we present the design of a visual XSLT programming tool called Vishnu. The Vishnu application aims to generate XSLT transformations from pairs of related documents, corresponding to source and target documents of an XSLT transformation.

3.1 Architecture

The architecture of the Vishnu application is divided in two parts - the Editor and the Engine - as depicted in Figure 1. It includes the following components:

- **the Editor** is an Eclipse plug-in that loads a pair of XML documents, respectively, a source and a target document of an XSLT transformation. The Editor uses the Vishnu API to interact with the Vishnu Engine (e.g. set of the imported documents, set a new map, generate the XSLT program);
- **the Mapper** maintains an XML map file identifying the correspondences between the two documents. These identifications can be inferred automatically by the Mapper or manually set through the Editor;
- **the Generator** uses a second order transformation to generate a specific XSLT program, based on the correspondences set by the Mapper;
- **the Refiner** receives the previous XSLT file and uses it as input for a serie of transformations. These transformations aim to refine the previously achieved XSLT program. The final result would be a generic XSLT file representing a more structured and generic transformation between any document instances of the respective types;
- **the Cleaner** receives the final transformation produced by the Refiner and replaces XSLT instructions by corresponding constructions in the target language. For instance, attributes constructed with the `xsl:attribute` element and the `xsl:value-of` are replaced by an attribute-value pair with the XPath expressions surrounded by brackets.

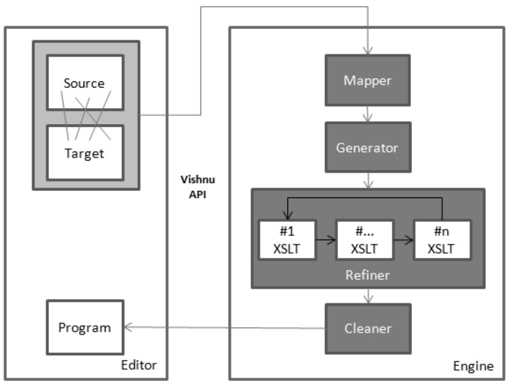


Fig. 1. Vishnu architecture.

In the following subsections we detail the internal components of Vishnu.

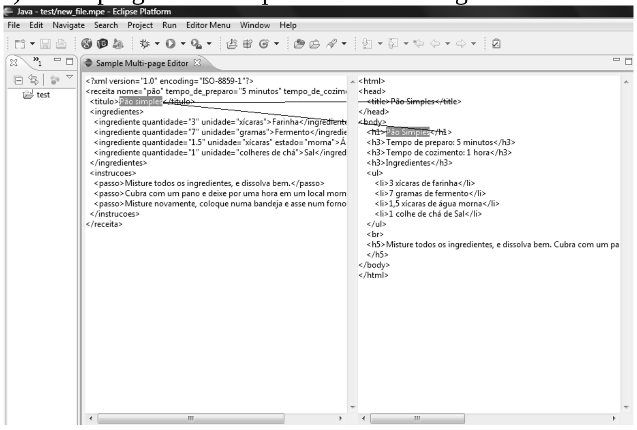
3.2 Editing

The front-end of Vishnu is an Eclipse plug-in. In this plug-in the "programmer" edits a pair of XML documents as examples of source and a target documents for the intended XSLT transformation. These XML instances may be created:

- from scratch, using the two XML Editors included in the GUI;
- guided by their type definitions, using the Eclipse completion mechanism.

In the last approach the built-in XML instance generator receives a schema file and automatically generates a valid instance. The user can also define several configuration options such as create optional elements/attributes, create a first choice of a required choice or even fill elements and attributes with data.

Regardless of the choice, the user can identify the correspondences between the two documents by clicking in the source and target text, respectively. The Editor component draws a line connecting both texts. A scenario of the graphical user interface (GUI) of the plug-in for Eclipse is shown in Figure 2.


Java - test/new_file.mpe - Eclipse Platform
File Edit Navigate Search Project Run Editor Menu Window Help
test
Sample Multi-page Editor

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<receita nome="pão" tempo_de_preparo="5 minutos" tempo_de_cozim
  <titulo>Pão simples</titulo>
  <ingredientes>
    <ingrediente quantidade="3" unidade="xícaras">Farinha</ingrediente
    <ingrediente quantidade="7" unidade="gramas">Fermento</ingredie
    <ingrediente quantidade="1.5" unidade="xícaras" estado="morna">Á
    <ingrediente quantidade="1" unidade="colheres de chá">Sal</ingred
  </ingredientes>
  <instrucoes>
    <passo>Misture todos os ingredientes, e dissolva bem.</passo>
    <passo>Cubra com um pano e deixe por uma hora em um local morn
    <passo>Misture novamente, coloque numa bandeja e asse num forno
  </instrucoes>
</receita>
```

```
<html>
<head>
  <title>Pão Simples</title>
</head>
<body>
  <h1>Pão Simples</h1>
  <h3>Tempo de preparo: 5 minutos</h3>
  <h3>Tempo de cozimento: 1 hora</h3>
  <h3>Ingredientes</h3>
  <ul>
    <li>3 xícaras de farinha</li>
    <li>7 gramas de fermento</li>
    <li>1,5 xícaras de água morna</li>
    <li>1 colhe de chá de Sal</li>
  </ul>
  <br>
  <h5>Misture todos os ingredientes, e dissolva bem. Cubra com um pa
  </h5>
</body>
</html>
```

Fig. 2. The Vishnu Eclipse plug-in.

The GUI will include two side-by-side windows for editing respectively the source and target transformations. The widgets of these windows support XML editing for highlighting the XML tags and enable completion based on DTD or XML Schema, if declared for the document.

The programmer is able to pair contents on these windows by drawing links where the origin is on the source document and the destination is on the target window. Origin and destination must be character data, either text nodes or attribute values. As can be seen in the example, this correspondence in not a mapping since the same text on the source document may be used in several points of the target document.

3.3 Pairing

Correspondences can be set manually through the Editor GUI or inferred by the Engine. When in automatic pairing mode Vishnu tries to identify pairs based on:

- Text matches (text or attribute nodes);
- Text aggregation.

In the first mode strings occurring on text and attribute type nodes on the source document are searched on the text and attribute nodes of the target document, and only exact matches are considered. In this mode a single occurrence of a string in the source document may be paired with several occurrences in the target document, as depicted in Figure 3.

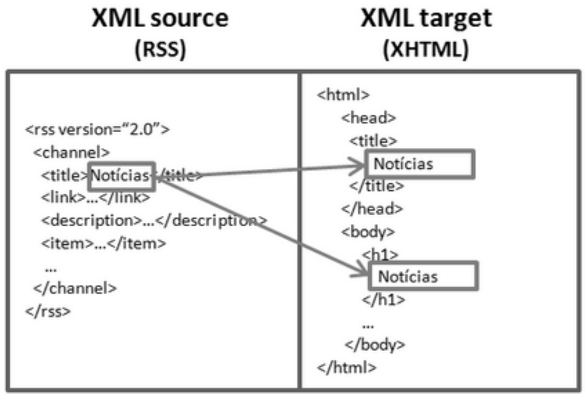


Fig. 3. Automatic mapping - exact match between single texts.

In the second mode Vishnu tries to aggregate strings in the source document to create a string in the target document. In Figure 4 we illustrate with a simple case where 3 strings occurring in attributes and text nodes can be concatenated into a part of the text node on the target document. In this mode several strings on the source document can be paired with strings on the target document.

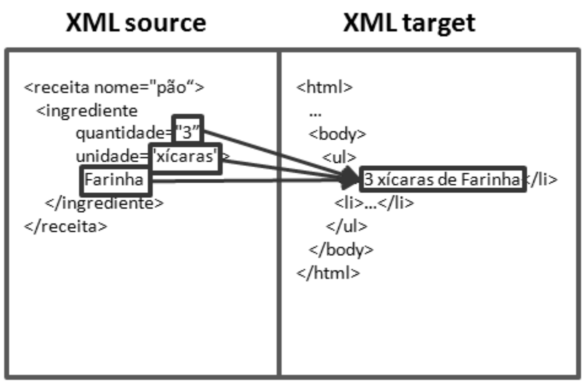


Fig. 4. Automatic mapping - subset of aggregation of texts.

After automatic pairing, the inferred correspondences are presented in the GUI with lines connecting the two XML documents. The user can then manually reconstruct the pairing of string between both documents, as explained in the previous sub-section.

The result of pairing the examples is a document including the actual documents and a list of pairs of XPath expressions relating them. This document is formally defined by an XML schema depicted diagrammatically in Figure 5.

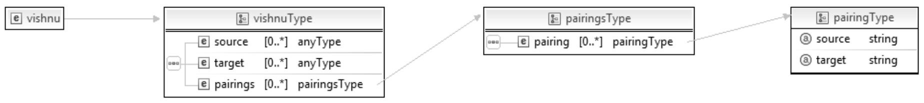


Fig. 5. The pairing XML language.

The pairing XML language has an element called `vishnu` as the root element with three top elements:

- `source` - a copy of the source document;
- `target` - a copy of the target document;
- `pairings` - list of pairing relating the two documents.

Each correspondence is defined by a `pairing` element with two attributes for selecting textual occurrences in both documents: source and target. The source attribute includes a valid XPath expression selecting the text to pair in the source document. The target attribute includes a valid XPath expression selecting the text of the target document. Based on the example of Figure 3, we present the correspondent XML pairing language:

```
<vishnu xmlns="http://www.dcc.fc.up.pt/vishnu">
  <source>
   <rss version="2.0" xmlns="http://backend.userland.com/rss2"/>
     <channel>
       <title>Notícias</title>
       <link>…</link>
       <description>…</description>
       <item>
```

```
        …
        </item>
      </channel>
    </rss>
  </source>
  <target>
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
        <title>Notícias</title>
      </head>
      <body>
        <h1>Notícias</h1>
         …
      </body>
    </html>
  </target>
  <pairings>
      <pairing
         source = "/rss[1]/channel[1]/title[1]/text()"
         target = "/html[1]/head[1]/title[1]/text()"/>
      <pairing
         source = "/rss[1]/channel[1]/title[1]/text()"
         target = "/html[1]/body[1]/h1[1]/text()"/>
   </pairings>
</vishnu>
```

3.4 Generating

The Generator component is responsible for the generation of a specific XSLT program based on a given pairing. The component receives as input a document in the paring language introduced in the previous section and, using a second order transformation, produces a specific XSLT program. This program is already a transformation that applied to the source document produces the target document, but is too specific and almost illegible.

The initial program produced by the generator has a single template. The generator iterates over all elements in `//vishnu/target` elements while checking if its absolute path corresponds to any pairing defined in the `//pairing/@target` attributes. In case of correspondence, it includes a `value-of` XSLT element with the respective `//pairing/@source` attribute, otherwise it just copies the element.

As an illustration we present the output of this second order stylesheet based on the example included in the previous subsection.

```
<xsl:template match="/">
  <html>
    <head>
      <title>
        <xsl:value-of
select="/vishnu/source/rss[1]/channel[1]/title[1]/text()"/>
      </title>
    </head>
    <body>
      <h1>
        <xsl:value-of
select="/vishnu/source/rss[1]/channel[1]/title[1]/text()"/>
      </h1>
      …
    </body>
  </html>
</xsl:template>
```

3.5 Refining

The goal of the Refiner component is to produce a high quality XSLT program from the initial program received by the Generator. The refinement of the program is achieved through the application of a set of second order transformations - called refinements - that simplify and generalize the initial program. These second order transformations act as rewrite rules of a rewrite process that normalizes the initial program.

Each refinement introduces either a small change to the XSLT program or preserves it as it was. The refinement algorithm is rather simple: the set of refinements is repeatability applied to the program until it converges. Convergence occurs when none of the available refinements is able to introduce a modification.

The control of the refinement process is implemented in Java, rather than in XSLT. This separation encourages the modularity and reusability of the refinement transformations which would be harder to achieve if the whole refinement process was encoded in a single XSLT. With this approach is easy to introduce new refinements or to temporarily switch them off. It is easier to change a single and simple XSLT file than to change the code and recompile the application. There are two types of refinements - simplifications and abstractions – that are presented in the following sub-subsections.

3.5.1 Simplifications

Simplifications are refinements that preserve the semantics of the program while changing its syntax. Preserving the semantics means that, for all documents S and T, if a program P transforms document S in document T then the program P', resulting from a simplification refinement, will also transform S to T.

Simplifications can be used for different purposes. They can be used to improve the readability of XPath expressions or to extract global variables. The following paragraphs illustrate this concept with concrete simplifications and examples of the refinements they introduce.

- **Context:** extracts the common prefix of all the XPath expressions from `value-of` elements in the same template and append it as a suffix of the `match` attribute on the `template` element.

Table 1. The Context stylesheet.

Source XSLT	Result XSLT
<xsl:template match="a"> ... *...<xsl:value-of select="b/c"/>* *...<xsl:value-of select="b/d"/>* *</xsl:template>*	*<xsl:template match="a/b"> ...* *...<xsl:value-of select="c"/>* *...<xsl:value-of select="d"/>* *</xsl:template>*

- **Melt:** two or more templates with the same containers are merged into one in which the `match` attribute is an expression that combines the terms of the original attributes match using the operator (|) that computes two or more node-sets.

Table 2. The Melt stylesheet.

Source XSLT	Result XSLT
<xsl:template match="a"> ... *</xsl:template>* *<xsl:template match="b"> ...* *</xsl:template>*	*<xsl:template match="a | b"> ...* *</xsl:template>*

- **Extract:** strings inside the templates are assigned to global variables.

Table 3. The Extract stylesheet.

Source XSLT	Result XSLT
<xsl:template ...> xpto </xsl:template>	*<xsl:variable name="x" select="'xpto'"/>* *...* *<xsl:template ...>* *<xsl:value-of select="$x">* *</xsl:template>*

- **Join:** different variables within the same scope and the same content are merged into a single variable.

Table 4. The Join stylesheet.

Source XSLT	Result XSLT
<xsl:variable name="x1" select="'xpto'"/> *<xsl:variable name="x2" select="'xpto'"/>*	*<xsl:variable name="x1" select="'xpto'"/>*

3.5.2 Abstractions

Abstractions are refinements that change both the syntax and the semantics of the program, although the refined program must retain the intended semantics of the example documents. This means that, for the documents S and T given as example, if a program P transforms document S in document T then the program P', resulting from a abstraction refinement, will also transform S to T.

Abstractions can be used for different purposes. For instance, they can be used to generalize templates and to restructure large templates in several smaller ones. The following paragraphs illustrate this concept with concrete abstractions and examples of the refinements they introduce:

- **Generalize:** two or more templates with the same container and a `match` attribute differing only in the "index" are merged into one and is removed the last predicate of the attribute `match`.

Table 5. The Generalize stylesheet.

Source XSLT	Result XSLT
<xsl:template match="a[1]"> ... </xsl:template> *<xsl:template match="a[2]"> ... </xsl:template>* *<xsl:template match="a[3]"> ... </xsl:template>*	*<xsl:template match="a"> ... </xsl:template>*

- **Structure:** fragment templates that contain XPath expressions with a common prefix.

Table 6. The Structure stylesheet.

Source XSLT	Result XSLT

<xsl:template match="a">
  <X>
    <xsl:value-of select="b/x">
    <xsl:value-of select="b/y">
  </X>
  <xsl:value-of select="c">
</xsl:template>
	<xsl:template match="a">
	  <xsl:apply-templates select="b"/>
	...
	  <xsl:value-of select="c">
	</xsl:template>
	
	<xsl:template match="b">
	  <X>
	    <xsl:value-of select="x">
	    <xsl:value-of select="y">
	  </X>
	</xsl:template>

3.6 The Vishnu API

Vishnu was conceived as an interactive tool integrated in Eclipse. Nevertheless, it was designed as two autonomous components: the editor and the engine. The editor is an Eclipse plug-in and concentrates all the tasks related with user interaction and integration with other Eclipse tools. The engine concentrates all the tasks related with the automatic creation of an XSLT program from examples using second order transformations. The communication between these two components is regulated by the Vishnu API.

By separating concerns in these two components we enable the non-interactive use of Vishnu. The engine has a command line interface to create XSLT programs from example files. Using Vishnu in this mode is as simple as executing the following command line.

```
$ java vishnu.jar source.xml target.xml > program.xsl
```

The Vishnu engine can also be invoked from other Java programs through the Vishnu API. This API may be used to create new user interfaces for Vishnu. For instance, a web interface based on the Google Web Toolkit (GWT) or a Swing based desktop interface. In general Vishnu may used by any application needing to create XSL transformations from examples. Java programs using the API must instantiate the engine using the static method *Engine.getEngine()* and use the following methods exposed by the Vishnu API:

`void setSource(Document source)`
Set source document example for the intended transformation.
`Document getSource()`
Get given source document example for the intended transformation .
`void setTarget(Document target)`
Set target document example for the intended transformation.
`Document getTarget()`
Get given example of target document for the intended transformation.
`void resetPairings()`
Reset all previously defined pairings.
`void addPairing(String exprSource, String exprTarget)`
Add a pair of XPath location respectively on the source and target documents.
`List<Pair> getPairings()`
Returns the list of pairings.
`void inferPairings()`
Infer pairings from the given source and target documents.
`Document program()`
Produce a XSLT program from the examples and their pairings.
`Set<String> getFeatureNames()`
Return a list of names of features controlling the refinement process.
`public boolean getFeature(String name)`
Get the given feature status.
`public void setFeature(String name, boolean value)`
Set the given feature status.

4 Conclusions and Future Work

We presented the design of Vishnu - a visual XSLT programming tool for Eclipse based on examples. Visual XSLT programming in Vishnu is based on drawing correspondences on examples of source and target documents. This data is fed to a generator that produces an illegible and over specialized XSLT program. The initial program is then rewritten into a more legible and general XSLT program using a set of elementary second order transformations called refinements. When no more refinements can be introduced the process stops and the last version of the program is cleaned up. This final version is then presented to the programmer on the user's interface of Vishnu.

The Vishnu project is in the design phase. At this stage we have a design and a prototype implementation of the engine. The generator is already implemented and XSLT programs are produced from examples. The main part of this project - creating the second order transformations to process the program - is just starting. We identified the main types of refinements - simplifications and abstractions - and examples of transformations of each type. We are currently writing a library of templates to support the development of refinements.

Developing a good set of refinements will be a challenge in itself. Proving that a particular set of refinements is confluent may be an even harder task. Confluence is an important property for the rewrite process to ensure its termination and to create a normal form for XSLT programs. A confluent set of refinements would open the use of the refiner to any XSLT program and not just to those produced by the generator. The refiner would be a tool in itself and could be used to refactor XSLT programs within an XSLT editor.

Vishnu will incorporate also a user interface as an Eclipse plug-in. Currently the plug-in prototype is less mature than the engine. The main challenge is editing graphical primitives, such as lines, across XML editing widgets. We plan also to experiment with web interfaces for developing XSLT transformations based on GWT.

References

1. Stylus Studio - http://www.stylusstudio.com/
2. Altova StyleVision - http://www.altova.com/stylevision.html
3. Tiger XSLT Mapper - http://www.axizon.com/
4. XSL Tools - http://marketplace.eclipse.org/content/xsl-tools
5. oXygen - http://www.oxygenxml.com/eclipse_plugin.html
6. XMLSpy Eclipse editor - http://www.altova.com/xmlspy/eclipse-xml-editor.html
7. OrangevoltXSLT - http://eclipsexslt.sourceforge.net/
8. X-Assist - http://sourceforge.net/projects/x-assist/
9. Dexter-xsl - http://code.google.com/p/dexter-xsl/
10. VXT: A Visual Approach to XML Transformations. Emmanuel Pietriga, Jean-Yves Vion-Dury and Vincent Quint. Proceedings of the 2001 ACM Symposium on Document engineering, USA
11. FOA. Formatting Objects Authoring tool - http://foa.sourceforge.net